RESEARCH ARTICLE

Customisable and scalable automated assessment of C/C++ programming assignments

Pedro Delgado-Pérez[1] | **Inmaculada Medina-Bulo**[1]

[1]Department of Computer Science and Engineering, Universidad de Cádiz, Puerto Real, Spain

Correspondence
Pedro Delgado-Pérez, Department of Computer Science and Engineering, Universidad de Cádiz, Puerto Real, Spain
Email: pedro.delgado@uca.es

Funding information
European Commission (FEDER); Spanish Ministry of Science, Innovation and Universities: project FAME (RTI2018-093608-B-C33); University of Cádiz: innovation projects sol-201500054192-tra and sol-201600064680-tra

The correction of exercises in programming courses is a laborious task that has traditionally been performed in a manual way. This situation, in turn, delays the access by students to feedback that can contribute significantly to their training as future professionals. Over the years, several approaches have been proposed to automate the assessment of students' programs. Static analysis is a known technique that can partially simulate the process of manual code review performed by lecturers. As such, it is a plausible option to assess whether students' solutions meet the requirements imposed on the assignments. However, implementing a personalised analysis beyond the rules included in existing tools may be a complex task for the lecturer without a mechanism that guides the work. In this paper, we present a method to provide automated and specific feedback to immediately inform students about their mistakes in programming courses. To that end, we developed the CAC++ library, which enables constructing tailored static analysis programs for C/C++ practices. The library allows for a great flexibility and personalisation of verifications to adjust them to each particular task, overcoming the limitations of most of the existing assessment tools. Our approach to providing specific feedback has been evaluated for a period of three academic years in a course related to object-oriented programming. The library allowed lecturers to reduce the size of the static analysis programs de-

veloped for this course. During this period, the academic results improved and undergraduates positively valued the aid offered when undertaking the implementation of assignments.

KEYWORDS
Automated assessment, computer programming, programming courses, computer science education, static analysis

1 | INTRODUCTION

The activity of programming is key to computer scientists. Acquiring this ability requires training with lots of exercises to obtain the necessary skills. However, the correction of programming tasks performed by lecturers is known to be a time-consuming, tedious and even error-prone task [1]. This correction implies that the lecturer has to visually check for non-complying elements within students' programs. An additional issue is the time elapsed between assignment submission and the moment when corrections are returned to students. This means that, despite the importance of providing instant assessment [2, 3], students do not always receive immediate feedback about their developments. Therefore, finding novel methods that help lecturers in this labour can facilitate learning and evaluation in courses where programming skills are taught.

Context In recent years, several approaches supported by technology have been proposed towards the automation of the assessment process and the improvement of the learning path in general [4, 5]. While the assessment of students' solutions can be based on diverse criteria [6], the design of test suites to check their functionality is a widely-used technique. Still, there are several requirements that cannot be verified when the program is executed (known as dynamic analysis), mainly with respect to the use of good programming habits or the programming style [7]. For instance, as pointed out by Horstmann and Budd [8], choosing descriptive names makes the code much more readable and facilitates maintainability.

As a motivating example, consider the following requirement extracted from the statement of a programming exercise:

> ***Requirement 1:*** *Provide a redefinition for operators == and ! = to compare two objects of class Date. To that end, first define operator == and then reuse that operator in the definition of operator !=.*

The idea behind this requirement is that the student becomes accustomed to reusing previous definitions instead of duplicating the same functionality (a threat to maintainability). In this case, the expected solution should be as follows:

```
//Two dates are equal if they share the same day, month and year.
bool operator ==(const Date& d1, const Date& d2)
{
   return d1.day() == d2.day() && d1.month() == d2.month() &&  d1.year() == d2.year();
}
```

```
bool operator !=(const Date& d1, const Date& d2)
{
   return !(d1 == d2);
}
```

As indicated in the requirement, the definition of *operator* != is based on the definition of *operator* ==. However, by executing a test suite, it is not possible to ensure if a student reused *operator* == in the definition of *operator* !=. The only way to check whether requirement 1 is met or not is to perform a code review in order to identify a call to *operator* == within the definition of *operator* !=. As it can be deduced, there is a great variety of aspects beyond program functionality that deserve special attention in order to prepare competent programmers.

An outstanding alternative is the use of *static analysis* [9]. This technique has the potential to embody the manual code review performed by lecturers in an automated way. As such, static analysis offers the possibility to automatically verify the compliance of students' solutions with the conditions established in each specific task. This analysis is based on different rules that the developed code should satisfy. Indeed, the above example illustrates the situations that static analysis can contribute to verify.

Problem Despite the appealing benefits derived from the application of static analysis to students' solutions, lecturers and students may find significant constraints when using existing tools developed to this purpose, especially for C/C++. On the one hand, most of these tools provide a fixed set of verifications and metrics. This means that they are hardly customisable to each particular practice beyond enabling/disabling the rules or changing their associated scores, like in Style++ [10]; this also means that users cannot extend these tools with new verifications overall. Moreover, these verifications should be configurable so that they only apply to specific code elements (e.g., we may want to apply a verification only to a single class). Furthermore, different lecturers might want to follow different criteria based on their personal experience. In general, there is a lack of flexibility and lecturers cannot adapt most of the existing tools to their own needs, as pointed out in a recent survey [11]. Therefore, a personalised list of verifications for each assignment is required. Moreover, it is still unclear the impact that feedback derived from a more fine-grained code analysis could have on students learning. On the other hand, the implementation of rules to check the fulfilment of certain requirements may be difficult for any lecturer if this activity has to be coded starting from scratch. Therefore, a procedure that eases and guides the development of tailored assessment programs is required.

Contribution This paper presents the *CAC++ library (Code Assessment of C/C++ programs)*, which aims to simplify both the work of lecturers and the learning of students, thereby alleviating the aforementioned issues. The library allows lecturers to design personalised assessment programs for C/C++ exercises (called *check programs*) and reuse the verifications, customising them to different practices. By personalised check programs, we mean that we can use a completely different set of verifications to statically analyse each of the assignments, according to the difficulty of the exercise and our preferences. Also, we mean that we can associate those verifications with specific elements of the code so that the checks only come into play at appropriate times. In this way, these programs can generate specific feedback based on the configured verifications, providing more accurate messages that can be adapted to the level of knowledge of our students. At the same time, the library has been prepared to be intuitively extended with new verifications. The main contributions of this paper are:

- **We present a novel automated method to assess programming tasks and provide feedback that is specific to each assignment**. Our approach overcomes limitations of current tools, which lack in providing flexibility to lecturers

and appropriate mechanisms to create new check programs tailored to each programming exercise. The paper aims to show the extent of the approach and to assess the benefits of its application, serving as starting point for new contributions in the future towards this direction.

- **We present the CAC++ library, which acts as a repository of verifications to derive new C/C++ check programs**. The power of CAC++ resides in three aspects: (1) unlike most of the existing tools, it is possible to adjust each verification to each particular situation; this means that each check will be triggered where expected (and not everywhere). Additional functionalities, such as the use of regular expressions and wild cards, increase expressiveness; (2) the library is based on a mature open-source compiler, *Clang* [12], which provides a robust and comprehensive way to analyse the Abstract Syntax Tree [13] or AST (i.e., an intermediate representation of the code generated by the compiler); (3) the library includes several verifications, but it has been devised scalable because there is a countless number of possible verifications. As such, maintainability and extensibility, often disregarded, become key aspects taking into account that programming exercises are constantly updated. To that end, CAC++ has been internally divided into well-defined modules. This helps the user deal with the different steps that static analysis entails.
- **We evaluate the impact of providing specific feedback to students in programming courses**. This was achieved by putting into practice this innovation over the last three academic years in a course about the object-oriented paradigm in the second year of the Degree of Computer Science Engineering in our University. The developed check programs were provided to the students to help them complete each assignment. The results give evidence that there was an enhancement in the academic achievement with statistical significance in that period. According to a survey, students found useful working with check programs overall, which also increased their awareness about the need for testing their code. The use of the library can greatly reduce the size of check programs, and therefore the effort required to develop them. We found that these check programs can provide useful and timely feedback to students and bridge the gap between students and lecturers, as the error/suggestion messages become an interesting point of communication.

The structure of the paper is as follows. Section 2 exposes the difference between static and dynamic analysis, and addresses existing approaches to improve the teaching of programming skills. Section 3 motivates the development of a new method based on static analysis for this purpose, and then describes the features of the library and its structure, supported by a running example. Section 4 deepens in the library by presenting real examples of application and limitations. Section 5 shows the results of the evaluation (effort required in terms of check program's size, academic result and survey to students) and discusses lessons learned. Finally, the last section presents conclusions and future work.

2 | BACKGROUND

2.1 | Code assessment and static analysis

The process of correction in programming practices has traditionally been a troublesome task because of its arduousness and the time required to revise all submissions of all students enrolled in a course. Usually, lecturers need to manually inspect students' programs. The goal is to ensure that students satisfy the conditions imposed on the assignments or apply a good programming style. This process of revision is generally termed *code review*.

Static analysis [9] partially simulates the process of code review by automatically analysing the code of a program. This is achieved by previously defining a set of rules that verify that proper coding conventions are used. On the basis

of those rules, the code is *read* or traversed, informing about non-complying parts of the code.

According to Naik and Tripathy [7] and Goedicke et al. [14], static and dynamic analysis are two complementary phases of unit testing. Static analysis is different from dynamic analysis, as no execution of the program is needed, and both types of analysis pursue different objectives. While static analysis seeks to verify that the code follows certain coding standards or detect potentially problematic parts of the code, dynamic analysis checks that the program behaves as expected through the test suite execution. Consequently, any program should undergo both kinds of tests, as pointed out by Naik and Tripathy [7]. Going back to our motivating example (see requirement 1 in Section 1), the goal of a test suite should be to reveal possible errors in the comparison performed by *operator* !=. Static analysis, however, could check that the functionality is not duplicate (as requested by the requirement).

2.2 | Related work: computer-aided assessment

The computer-aided assessment of programming practices is supported by many and diverse tools developed over the last years. A fairly comprehensive list of existing tools can be seen in the review by Pettit et al. [4]. In that review, the authors analyse the usefulness of those tools in the code assessment based on data found in the literature. Other previous surveys [15, 16] evaluate and compare the features of different assessment tools. A more specific review by Striewe and Goedicke [5] collects different static analysis approaches for assessing programming tasks. Ceilidh [17] is one of the first and most famous systems to support teaching of computer programming for C. This tool allowed to perform both static analysis (such as to check the proper use of comments and indentation, or to calculate complexity and readability metrics) and dynamic analysis (through output matching in the execution of data sets). These tools have been increasingly refined, turning coarse pattern matching into more fine-grained code analysis. Most of these tools define a fixed set of rules that can be applied for the assessment, which in general cannot be extended with new verifications. Overall, they can only be configured to enable or disable some rules, or change the importance of each verification. The tool JACK, however, allows for the execution of specific checks for the assessment of Java programs [18]. These verifications are described by a specification language based on abstract syntax graphs, which provide a structured representation of the Java source code [14]. In the library presented in this paper, we follow a similar approach to JACK in order to achieve a customisable and scalable tool for C/C++ programming practices. In our case, we propose an open library that acts as a repository of verifications and that has been designed to be intuitively applied and extended with new checks. These verifications are implemented through pattern matching on the AST generated by *Clang*. This is a well-established compiler, which allows performing a robust static analysis of students' programs. Moreover, this pattern matching is supported by additional functionalities, such as the possibility to use regular expressions or wild cards.

Early experiences about computer-based assessment using Ceilidh were described in 1993 by Benford et al. [19]. Since then, many other similar tools have been developed to assist the teacher in programming practices for diverse languages, such as ASSYST for Ada [20], CAP for Pascal programs [21], Scheme-robo for the functional language Scheme [22], LinuxGym to test scripts [23], APOGEE for web-based computing [24] or JavaMarker for Java [25]. Ceilidh, later renamed CourseMarker [26] was also redesigned to support new languages like Java, multiple courses and interfaces, remote learning and achieve a more user-friendly and usable system. These tools introduce some other interesting features, such as the detection of possible plagiarism [22], online submission [27] or the assessment of the program efficiency. Ala-Mutka et al. [10] developed Sytle++ for the program style assessment in C++ programming courses. Unlike previous tools developed for C, this one was able to address more intricate structures related to the object-oriented paradigm. This tool takes as its basis some programming style guides for C++, such as the work by Meyers [28] or the industrial guidelines by Ellemtel laboratories [29]. Sytle++ also follows the work by Dromey [30]

to define proper measurements that correspond to software quality attributes (e.g., modularity and reliability). During the last years, some other tools have been constructed to address C++ practical exercises, such as CloudCoder [31] or CodeAssessor [32]. In our work, we devise a library for the assessment of C/C++ programs by reusing the API of *Clang*, which guarantees full coverage of the language features, including the new and recent C++ standards.

Most of these tools have been harnessed to award marks to students' solutions, evaluating the programming style and using different quality metrics (for instance, depth of inheritance or number of global variables [25]) and test cases, among others [6]. Also, PMD [33] or SonarQube [34] are examples of well-known general-purpose tools to unveil potential risks and provide metrics regarding code quality in different languages in an automated way. On the contrary, there are some works promoting semi-automatic approaches that merge the computer-based assessment and human judgement for a more complete evaluation, such as the works by Jackson [35] and Kalogeropoulos et al. [1] (e.g., it is not easy to mechanise whether the content of comments is meaningful). The study by Laakso et al. [36] also suggests that some exercises are best suited for face-to-face sessions; as a result, automatic and traditional assessment should be combined. In the experience reported in this paper, we introduce the use of check programs as a method to guide students when developing their programs instead of for evaluation purposes. In this way, students can practice the exercises in an autonomous manner and receive feedback continuously, as in the study by Restrepo et al. [37]. Regarding the feedback, a recent systematic literature review by Keuning et al. [11] focuses on tools that provide automated feedback and analyses the different types of feedback generated. In that review, the authors conclude that lecturers cannot easily adapt the tools and influence the feedback, and that increasing the level of detail in the generated messages would contribute to giving more effective feedback. Our library allows setting personalised messages for each particular situation thanks to the possibility of customising each verification, and we evaluate in this paper the effect of providing more specific feedback.

The related works by Clarke et al. [38] and Clegg et al. [39] seek to enhance learning by integrating software testing concepts into software engineering and programming courses, while Lemos et al. [40] analyse the benefits of teaching software validation techniques on code reliability. In our work, we also introduce static and dynamic approaches to test a program in the programming practices, and evaluate the perception that students have about the testing phase.

3 | CAC++ LIBRARY

3.1 | Why a library?

As previously discussed, each programming task should be accompanied by a check program tailored to the requirements in its statement. However, building new programs without a baseline incurs several issues, especially with a view to the future. These are described below:

- **Reuse of verifications** The generation of unrelated check programs leads to verifications differently implemented. Even if the same verification is required in different exercises, these check programs might slightly differ; this happens when the implementation of the verification focuses on the particularities of the targeted elements, or when different lecturers develop these programs. This issue clearly hinders the traceability of such verifications and complicates their reuse in new practices in the future.

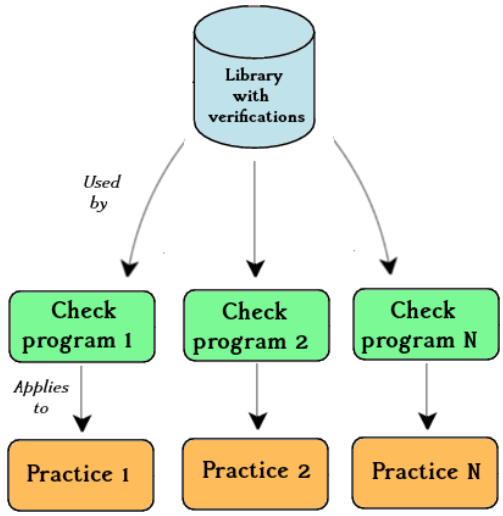

FIGURE 1 Diagram of application of static analysis supported by the CAC++ library. Each check program for each task is implemented following the library guidelines. All generated check programs draw upon the same set of verifications.

- **Ease of maintenance** Check programs developed without guidelines, in different points in time and by different lecturers, often result in programs of varied structure and low modularity and generalisation. This makes difficult their maintenance and to fix them if any issues are found.

- **Static analysis: a non-trivial task** Developing a program that performs static analysis typically requires reusing third-party libraries to this end. Especially in the case of general-purpose languages, the use of these libraries is not trivial. As a consequence, any lecturer can neglect to amend and update the verifications, especially if the adjustments are implemented from academic year to academic year.

The above commented aspects motivate the development of a mechanism that alleviates these issues and guarantees that assessment programs are regularly updated. Furthermore, C++ is increasingly growing as the new standards add many novel features [41]. A library containing the verifications and guiding their definition and application offers a suitable solution.

3.2 | Description

The CAC++ library has been designed to perform an automated and robust assessment of programming exercises in C/C++ applying static analysis to students' solutions. To that purpose, the library collects a set of parametrised verifications, which can be used to generate a check program tailored to each exercise. The novelty of this library lies in the following features:

1. **Personalised correction**: The library allows selecting which verifications are applied to check each practice, and adapting which elements of the code are exactly their target. Therefore, the library allows for a much finer control

with regard to the particular requirements in each assignment.

2. **Open and scalable**: In the context of coursework assessment, where exercises can change from one year to the next, maintainability and extensibility become two of the most important software quality attributes. Unlike other similar computer-aided assessment systems, whose code is closed, this library has been devised with a view to its extension with new verifications. As such, the lecturer can play the role of user and developer of the library at the same time thanks to its modular design.
3. **Underlying technology**: The library implements static analysis with the aid of a full-fledged compiler, *Clang* [12]. *Clang* provides a well-documented API that can be easily reused for our own purpose. In fact, this compiler was conceived to facilitate the design of new tools working at the source-code level. This translates into a one important advantage because of its simplicity and robustness. *Clang* libraries have been successfully integrated into other testing tools [42, 43].

The library has been developed with the aim to offer an intuitive but flexible method for its practical use. Figure 1 depicts the sequence of steps followed when using this library, where all check programs depend on the library. Therefore, any amendment to a verification in the library directly affects all check programs, that is, there is no need to fix each one individually.

Currently, the library counts with a set of verifications related to (among others):

- Fields, (e.g., to check the access level).
- Methods (e.g., to check whether the method is marked *noexcept*).
- Construction/destruction (e.g., to check the existence of certain constructors).
- Functions (e.g., to check whether a function is *friend* of a class).
- Miscellany (e.g., to check the inclusion of suitable headers).

The library can be freely downloaded from the web. `https://ucase.uca.es/cac`.

3.3 | Structure

The lecturer can make use of the library as a *user* (to create check programs for new practices) and as a *developer* (to define new verifications or extend existing ones):

- **As a user**, the library has been designed to facilitate the work of implementation of check programs, avoiding that the lecturer has to deal with the internal implementation.
- **As a developer**, the library presents a modular nature that helps distinguish and interpret the different parts that intervene in the execution of verifications. For instance, as it will be seen later on, all search patterns are gathered in the same module.

As such, the execution diagram can be separated into:

1. **Input**: Both, the *check file* designed by the lecturer for a specific task, and the *solution* implemented by the student form the input to the system.
2. **CAC++ Library**. The library comprises two main modules: the *controller* and the *executor* module. They make use of two supplementary modules: one that contains the *search patterns* and the other one to keep useful *information*

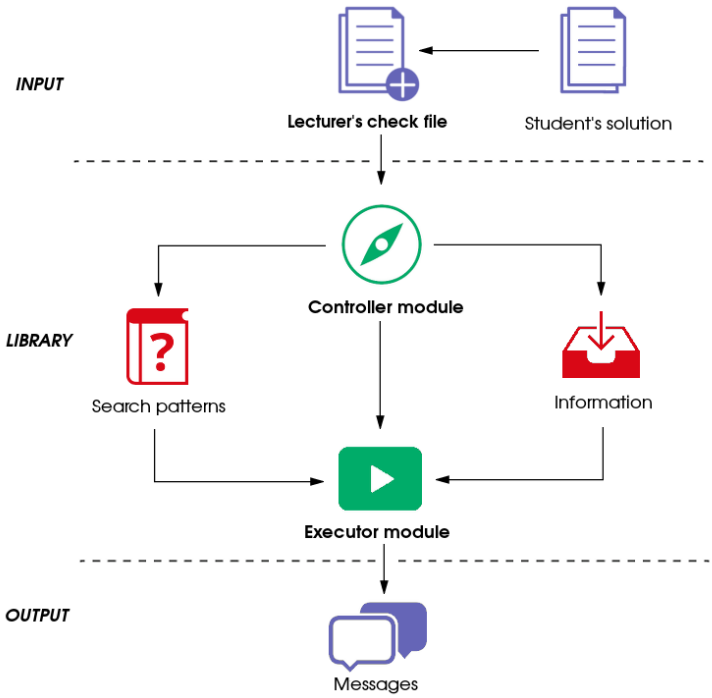

FIGURE 2 Workflow diagram when using the CAC++library.

to complete the verification.

3. **Output**. The output is the list of errors and suggestions reported after analysing the student's solution according to the requirements indicated by the lecturer.

Figure 2 depicts the workflow of the execution of a check program that uses the CAC++ library. Each element is described in detail below, supported by a running example.

3.3.1 | Input

Lecturer's check file

In this file, the lecturer indicates the verifications required to assess a task. The library includes an interface with a method to invoke each of the verifications. As a result, the check file is just a list of invocations to the library's methods. Appropriate arguments allow adjusting each verification to the particularities of each exercise.

Example

In our example, the lecturer desires to check whether the student's solution complies with the two following conditions in the class named *Example*:

1. The class has to include two (and only two) user-defined constructors.

```
#include "caclibrary.h"
#include <iostream>

int main(int argc, const char **argv){

   //Create an object to interact with the library
   checkCode c(argc, argv, "student.cpp", "Command: ./example student.cpp");

   c.setCorrectMessage("Class Example correct.");
   c.setIncorrectMessage("Review the messages related to class Example.");

   std::cout << "\nVerifications for the class Example" << std::endl;
   std::cout << "***********************************" << std::endl;

   //Check the existence of the class Example
   if(c.findClass({"Example"})){

      //Check that two constructors have been defined
      c.numberOfConstructors("Example", 2, false, "Review the information "
                             "about constructors in the practice.");

      //Check that mainMethod reuses referencedMethod (or referredMethod).
      c.methodWithReferencedMethod({"mainMethod"}, {{"int"}}, "Example", {"?"},
                                   {"refer.*edMethod"}, {{}}, "Example", {"?"},
                                   "Reuse is not being carried out as indicated in the statement.");

      //Apply verifications
      c.check();
   }
   else
       std::cout << "Class Example not found." << std::endl;
       std::cout << "***********************************" << std::endl;

       return 0;
  }
```

FIGURE 3 Check file for the running example.

2. The class has to include a method with the name *referencedMethod*, which is invoked within the method *mainMethod(int)*.

Figure 3 shows a possible implementation of a check file to apply these verifications. In this code:

- **findClass** (line 17) is used to check that there is a class with the name *Example*. Otherwise, the rest of verifications regarding that class are not executed.
- **numberOfConstructors** (line 20) is used to check condition (1). It receives (in order of appearance):
  - Name of the class.
  - Number of expected constructors.
  - Boolean to indicate whether the verification is *lenient* (i.e., a greater number of defined constructors is permitted) or not.
  - Personalised message.
- **methodWithReferencedMethod** (line 24) is used to check condition (2). This verification receives (in order of appearance):
  - Identification of the calling method: its name, types of the parameters, class and whether it is marked *const*.
  - Identification of the called method (same parameters as in the calling method). The use of a regular expression ("refer.*edMethod") will be later explained in Section 3.4.
  - Personalised message.

Students' solutions

The lecturer's check file is finally compiled together with the library to build the check program. Therefore, the student's program is the real input to the system. We should note that the student's solution does not require any modifications to be processed by the check program.

Example

According to the aforementioned conditions, the class in Figure 4 is a correct solution because the method *mainMethod* calls the method *referencedMethod*. Contrarily, the codes in Figure 5 are not correct because student A duplicates the code $a * 2$ inside *mainMethod* instead of calling *referencedMethod*, and student B provides a further constructor that is not required.

3.3.2 | Library modules

As mentioned previously, the library contains four modules (see Figure 2), which are described below:

Controller module

This module receives and processes the information provided by the lecturer. In brief, the information is prepared for the subsequent execution of static analysis in the executor module. As it can be seen in Figure 2, this module communicates with two supplementary modules:

- ***Search patterns***: This module collects all search patterns used to check each of the verifications. These patterns are based on existing expressions to analyse the code (called *matchers*) in the libraries of *Clang* [44]. At this point, the controller module fetches the appropriate patterns according to the verifications selected by the lecturer.

```
class Example{

    public:
       Example(): a(0) { }
       Example(int a_): a(a_) { }
       void mainMethod(int p){
          a = referencedMethod() + p;
       }
       int referencedMethod(){
          return a * 2;
       }
    private:
       int a;
};
```

FIGURE 4 Correct solution

```
// Student A:
class Example{

    public:
       Example(): a(0) { }
       Example(int a_): a(a_) { }
       void mainMethod(int p){
          a = a * 2 + p;
       }
       int referencedMethod(){
          return a * 2;
       }
    private:
       int a;
};
```

```
// Student B:
class Example{
      public:
       Example(): a(0) { }
       Example(int a_): a(a_) { }
       Example(int a1, int a2): a(a1 + a2) { }
       void mainMethod(int p){
            a = referencedMethod() + p;
       }
       int referencedMethod(){
           return a * 2;
       }
       private:
            int a;
       };
```

FIGURE 5 Incorrect solutions: student A - *mainMethod* should invoke *referencedMethod* (left); student B - there should be two constructors in the class instead of three (right).

Note that these patterns are parametrised. In this way, they can incorporate the data provided in the lecturer's check file.

Example

The pattern *classWithName_Matcher*, associated with the verification *findClass*, is shown in the following excerpt:

```
DeclarationMatcher classWithName_Matcher(string className){
```

```
    DeclarationMatcher matcher =
        cxxRecordDecl(
            matchesName(className)
        )

    return matcher;
}
```

As it can be seen, our pattern searches for a class (represented by the matcher *cxxRecordDecl*) that matches the name contained in *className*. This parameter is provided by the lecturer when invoking the verification. In our case, the name of the class is *Example* (see code in Figure 3).

- **Information**: This module records data provided by the lecturer. These data are necessary to complete the verification once the search patterns are executed.

  Example
  The verification *numberOfConstructors* checks whether the student has defined a specific number of constructors for a class. The pattern associated with this verification searches for all *constructors* within the class. Only after this pattern has been executed, the library is able to determine whether the number of constructors found is the one expected by the lecturer. Therefore, the parameters *number of expected constructors* and *boolean lenient* require being saved for that moment.

Executor module

This module is invoked with the method *check* (line 28) after all verifications have been set in the lecturer's check file (see Figures 2 and 3). Then, this module applies the list of verifications (previously prepared by the controller module) to the student's file. Internally, *Clang* derives an Abstract Syntax Tree (AST) [13] from the student's solution and the library traverses this AST in order to match the patterns [45] through the Clang API. The traversal of the AST is efficient because all verifications are applied at the same time, thus avoiding a new traversal of the code for each verification added. Each verification generates a result, which is later used to produce the output. Combining this solution with other programming techniques like reflection could be useful, although the reflection mechanism is not as flexible in C++ as in other languages.

3.3.3 | Output

Once the execution of the verifications has finalised, the last step is the display of results. This stage involves two elements: the results generated by each verification and the information previously saved, including the output messages set by the lecturer. For each verification that failed, its associated message is shown.

Example

If the student's solution corresponds to the code in Figure 4, the solution is correct. Therefore, only the message set with the method *setCorrectMessage* is shown (see line 10 in Figure 3):

```
Verifications for the class Example
**********************************

Class Example correct.
```

Contrarily, the application of this check program to the incorrect codes in Figure 5 will show the following messages to the students A and B, respectively. These messages are composed of the list of error/suggestion messages and a closing message set with *setIncorrectMessage*:

```
Verifications for the class Example
**********************************
Reuse is not being carried out as indicated in the practice.
-------------------
Review the messages related to class Example.
```

```
Verifications for the class Example
**********************************
Review the information about constructors in the practice.
-------------------
Review the messages related to class Example.
```

3.4 | Additional functionalities

The CAC++ library includes additional functionalities to increase flexibility in the application of verifications. It is worthy of mention:

- **Positive and negative case**: In most cases, the lecturer desires to check that certain elements have been included or used in the code. However, the lecturer may also want to check exactly the contrary, that is, that certain elements are not present in the code. Thus, several verifications are parametrised with a boolean to directly select the application of the positive or the negative case.
- **Wild card - ?**: A wild card, indicated by the ? symbol, can be used for a particular parameter instead of a fixed value. This is useful when some data are not known in advance or are unimportant. For instance, in the previous example, whether the involved methods were *const* was not relevant. As such, a wild card for that parameter was used instead.
- **Regular expressions**: A regular expression can be passed as argument of a verification instead of a fixed value. This is helpful when the same verification is expected to cover a set of entities, or to deal with the variability in the name of declarations. For instance, in the previous example, we used the regular expression *refer.*edMethod* in the verification *methodWithReferencedMethod* because the name for that method could be *referencedMethod* or *referredMethod*.

```
1
2  // Requirement 1:
3  c.functionWithReferencedFunction(
4                  {"operator!="}, {"const Date&", "const Date&"},
5                  {"operator=="}, {"const Date&", "const Date&"},
6                  "Revise the suggestions regarding reuse in relational operators."
7  );
8
9  // Requirement 2:
10 c.noExceptMethod(
11                 {"day", "month", "year"},
12                 {{""}, {""}, {""}},
13                 "Date",
14                 {"const", "const", "const"},
15                 "Suggestion: non-throwing methods should be marked 'noexcept'."
16 );
17
18 // Requirement 3:
19 c.deletedMethod(
20                {"CreditCard", "operator="},
21                {{"const CreditCard&"}, {"const CreditCard&"}},
22                "CreditCard",
23                "Revise the statement with respect to the copy of objects."
24 );
25
```

FIGURE 6 Invocations to the verification rules in the library to satisfy real requirements (1, 2 and 3).

4 | APPLICATION OF THE LIBRARY

In this section, we show examples of application of the CAC++ library and the limitations found when applying this approach.

4.1 | Real examples of application

To better understand the extent of the library and its expressiveness, we present several requirements extracted from programming exercises that students in our University have to solve (details on the subject and the students are later provided in Section 5). Figure 6 shows the instructions that we need to add to the check file in order to verify these requirements.

> ***Requirement 1:*** *Provide a redefinition for operators == and ! = to compare two objects of class Date. To that end, first define operator == and then reuse that operator in the definition of operator !=.*

Recall that our first requirement, presented in Section 1, pursues that *operator* != reuses the definition of *operator* == when comparing objects of type *Date*. The library's rule *functionWithReferencedFunction* is able to verify this condition. Figure 6 shows the invocation to this rule, whose format is similar to rule *methodWithReferencedMethod* (see

Section 3.3):

- Identification of the calling function (*operator* !=): its name and types of the parameters (line 4).
- Identification of the called function (*operator* ==): its name and types of the parameters (line 5).
- Personalised message (line 6).

> ***Requirement 2:*** *Do not forget to use the* noexcept *specifier to mark those methods in class* Date *that do not throw any exceptions.*

Namely, the *noexcept* specifier (included in the standard C++11) should be added to the methods *day*, *month* and *year*, as shown below:

```
int day   () const noexcept;
int month() const noexcept;
int year  () const noexcept;
```

This can be checked with the verification *noExceptMethod*. As it can be seen in Figure 6, the library allows concatenating a list of methods to avoid repeating three times the same instruction for each of the methods. Therefore, the verification can be invoked as follows:

- List of methods: *day*, *month* and *year* (line 11).
- List of types of the parameters in the specified methods (line 12), following the same order as the list of methods. Note that, given that these methods receive no arguments, the list of parameters is empty ({""}) in all three cases.
- Name of the class (line 13).
- *const* qualifier in the specified methods, again following the same order as methods (line 14).
- Personalised message (line 15).

> ***Requirement 3:*** *Class* Date *is used to represent the expiration date in class* CreditCard*. This class prohibits the copy of credit cards (there should not be two credit cards with the same number).*

To achieve this, students have to define as *delete* (standard C++11) the copy constructor and the overloading of the assignment operator of class *CreditCard*, as shown below:

```
CreditCard(const CreditCard&) = delete;
CreditCard& operator =(const CreditCard&) = delete;
```

The corresponding verification to check this condition is *deletedMethod*, which can be invoked with the following arguments:

- List of methods: copy constructor and overloading of the assignment operator (line 20).
- List of types of the parameters in the specified methods, following the same order as methods (line 21).

- Name of the class (line 22).
- Personalised message (line 23).

As it can be seen, the way in which the verifications are internally implemented in the library is completely transparent to the user. We should also remark that, for the sake of consistency, all the included verifications follow a similar format and structure, both externally when invoked as a user and internally in their implementation. This facilitates the labour of the lecturer, who can apply or implement verifications by analogy to other verifications. It also contributes to the homogenisation of the situations considered by each of the verifications. As an example, the AST contains implicit elements added by the compiler, such as the destructor for a class when the developer fails to define one; the analysis should be performed only on the code explicitly added by students, so all search patterns are configured to discard implicit elements.

4.2 | Limitations

Two main limitations were detected while building and applying these check programs. Namely:

- **Different solutions**: Sometimes, encompassing all possible situations in a verification is not easy. This can potentially lead to false positives or negatives. For instance, imagine that we want to verify whether the student has increased by one the value of the variable *counter* or not. To that end, we could implement a verification to search for the presence of the following expression:

  ```
  counter = counter + 1;
  ```

  However, among others, the following expressions are also valid:

  ```
  counter = VALUE_ONE + counter;          (being VALUE_ONE a constant)

  counter++;

  increaseCounter(1);
  ```

  Especially, in this last case, the verification becomes quite complicated as that implies the analysis of the called method.
- **Specific verifications**: There is a limitation inherent to static analysis when it comes to specific verifications. When the verification focuses on particular code elements, as *counter* in the previous point, a fixed name for elements is often required. This issue is especially relevant when we need to delimit the search because applying the verification to other elements of the same type would produce an undesired effect. As a consequence, this limitation forces us to establish the names of some elements beforehand in the statement. Therefore, those verifications will not take effect when students do not follow the instructions in their solution.

5 | EVALUATION AND DISCUSSION

5.1 | Goals and methodology

This section aims to evaluate and discuss the effect that the introduction of this innovation has on programming courses. To that end:

- The use of check programs has been applied in Object-Oriented Programming, a second-year subject in the Degree of Computer Science Engineering of our University. Students enrolled in this subject study C++ and should have completed several related subjects in this Degree, such as Introduction to Programming, Programming Methodology, and Algorithm Analysis and Data Structures, where they learn the C programming language.
- The innovation has been assessed in a period of three academic years since 2015 with 210 students in total. The set of students observed in each academic year was new except for repeating students. The course takes place over 15 weeks, with 2.5 theoretical hours and 2 practical hours per week.
- Students had to solve six assignments of increasing difficulty throughout the course. The set of programming tasks were the same in all three years, and students had between 1-3 weeks to submit their solutions to each task.

The lecturers assigned to this course developed check programs according to the statements of the exercises. Those check programs were transferred to the students together with the description of each task. Thus, the students could make use of them and work on the assignment until their code was compliant with the requirements set. We should note that test suites were also put at the disposal of the students as in previous years. These test suites could be applied to check the correct functionality of the solutions. Therefore, they had the opportunity to apply both static and dynamic unit testing. The innovation was assessed in terms of the following research questions (RQ):

- **RQ1: How much effort is required to develop check programs with the library?** To estimate this aspect, we analysed the size of the lecturer's check files developed for the practices of this course. We compared two different versions: with and without using the CAC++ library.
- **RQ2: What is the impact of the application of check programs on the academic results?** To this end, we compared the results in the academic years using the innovation with those in previous years, including a statistical study based on the marks.
- **RQ3: Do students find useful working with check programs?** As an additional indicator, it is relevant to know the students' perceptions of the use of these check programs. To do so, the students were surveyed about their satisfaction with the use of check programs at the end of the course and prior to the final exam. The same questionnaire was given to the set of students in each year.

5.2 | RQ1: Effort required

Lecturers of this subject designed six individual check programs for each of the six assignments that students had to face throughout the course. Each of these check programs implemented the specific verifications that each assignment required without the aid of the library. Once the library was developed, those check programs were translated into new versions that made use of the verifications contained in the library.

Table 1 provides an estimation of the effort required to create a new check program with the CAC++ library when compared to generating individual check programs. Namely, this table shows the number of lines of code of the six check programs with and without using the library.

As it can be seen from this table, the difference between both versions increases in general with the difficulty of the task (overall, the number of verifications required for an assignment is in line with its difficulty). This is because we can incorporate new checks by only adding personalised invocations to the corresponding verification rules in the library (see real examples in Section 4). Thanks to this, the developed check programs using the library gained in readability and also in maintainability in general. Measuring the effort required to integrate new verification rules into

TABLE 1 Size of *individual* check programs and check programs *using the library*, measured as lines of code.

Version	***Assignment***					
	1	2	3	4	5	6
Individual	432	522	682	443	941	1.203
Using the library	49	66	74	78	114	135

the library would merit further experiments.

5.3 | RQ2: Academic impact

This section aims to assess the impact of the innovation on the academic results in this subject. We should note that the programming practices are only taken into account to calculate the final qualification if students previously passed the final exam. Several factors can have influence on the overall results (e.g., the number of students in each year); according to this, we compared the academic results previous to the innovation with those derived from the years that encompass its application on average.

- **Success rate**: It is calculated as the ratio of *passing students* to the number of *students that attended the exam*. The objective of the success rate is to observe how many of the students that prepared the exam succeeded.
- **Academic performance rate**: It is calculated as the ratio of *passing students* to the number of *students enrolled in the course*. The objective of the performance rate is to observe how many of the students enrolled in the course ended up passing the exam. The performance rate is therefore equal or lower than the success rate because it includes those students that do not attend the exam.

TABLE 2 Academic results: success and performance rates, shown as percentage, divided by the three academic years and on average. *Previous* is the average result of these rates in the academic years prior to the application of the innovation.

Rate	***Academic year***				**Previous**
	1^{st}	2^{nd}	3^{rd}	Avg.	
Success	57.1	58.4	75.0	**63.5**	52.6
Performance	32.0	33.9	56.4	**40.8**	31.6

Table 2 compares the results in the academic years using the innovation and in previous years. Focusing on the average, both success and performance rates have notably increased over the last years. Indeed, all values in the three years with the innovation are over the average in previous years. The success rate increased by almost 11 per cent when compared to the mean result of the previous years (63.5% vs 52.6%). This improvement of the pass rate, however, might be caused simply by a decrease in the number of students that decide to attend the exam just to make an attempt. The academic performance shows that this is not the case: the percentage of students passing the exam with respect to the total number of students has also increased from 31.6% to 40.8%. It is remarkable that both rates have increased year by year. This might be related to an increasing acceptance of the innovation in the mechanics of the course, derived from a greater experience of its lecturers, and the subsequent amendments to the check programs.

Additionally, we wanted to know whether the differences observed in both periods were statistically significant taking into account the marks that students obtained in the exams during these years. To do so, we performed a statistical test comparing two independent samples: (1) all marks obtained by students in the years with the innovation and (2) all marks in the previous years. Namely, we run the Mann-Whitney U test between both groups in the statistical framework *R*. The result of Mann-Whitney U test (W = 57300; *p-value* = 0.028) leads us to accept the alternative hypothesis with 0.05 significance level, that is, that the median of the marks obtained in the previous years is statistically lower than the one in the years with the innovation.

TABLE 3 Survey's questions (SQ).

#	Question
SQ1	Did you find static analysis useful to complete the practices?
SQ2	Do you deem that you have learned more thanks to the verifications included in the check programs?
SQ3	Do you think that static analysis has somehow helped you improve your programming style?
SQ4	Do you consider that you have become more aware of the importance of the software testing phase through the use of check programs?
SQ5	Had it not been for static analysis, do you think that you would not have been aware of some of the improvement suggestions or that these would have gone unnoticed?
SQ6	After completing the practices of the course, do you consider static analysis a necessary complement to a test suite to verify that the requirements of the practices are met?
SQ7	Would you recommend the use of static analysis in other courses of the Degree?
SQ8	In general, how do you value this innovation in the course?
SQ9	Further comments and suggestions

5.4 | RQ3: Students' opinion

Table 3 collects the nine questions included in the survey to know about the students' satisfaction. Using a 5-point Likert scale, questions SQ1 to SQ8 could be assigned an integer value in the range 1 to 5 (being 1 and 5 the worst and best value, respectively). SQ9 was simply a free-text field for comments about the innovation. The students were informed that the survey was entirely optional and anonymous.

Table 4 shows the number of participants in the survey per academic year and in total. There were, however, some answers that presented one of the extreme values, 1 or 5, in all questions. The answers from those participants cannot be judged as reliable and they were therefore removed from the evaluation to avoid skewing results.

The survey's results are presented in Table 5. This table shows the percentage of participants that marked each of the values in each of the questions. This allows us to observe the most selected value in each question (i.e., the mode). The last column also shows the calculated average value.

As it can be seen, all questions were positively valued (average value greater than 3), with the value 4 and 5 as mode. Therefore, the innovation had a favourable reception from students despite the fact that it implies a greater effort to complete the coursework; notice also that the check programs play the role of revealing the defects in their code, which can be regarded as a burden for them. In fact, Ala-Mutka et al. [10] reported that they met with initial resistance when integrating automatic assessment into the course during the first year. A possible explanation for this difference is that we did not use the check programs for evaluation purposes but we provided them as an aid to complete the assignments. As such, students may feel more comfortable using these programs because they do not

feel the pressure of being constantly under evaluation.

TABLE 4 Number of participants in the survey divided by academic year: number of answers received, removed answers (those with all values 1 or 5) and total of answers considered for the evaluation.

Academic year	Answers	Removed	Total
1^{st}	17	3	14
2^{nd}	43	8	35
3^{rd}	46	4	42
Total	**106**	**15**	**91**

TABLE 5 Results of the survey's questions (SQ): percentage of answers of each value in the range 1 to 5 and average value (Avg.). The mode is highlighted with a box.

#Q	*Answer (%)*					Avg.
	1	2	3	4	5	
SQ1	6.6	14.3	18.7	29.7	30.8	**3.64**
SQ2	9.9	15.4	23.1	30.8	20.9	**3.37**
SQ3	11.0	12.1	24.2	30.8	22.0	**3.41**
SQ4	4.4	11.0	16.5	27.5	40.7	**3.89**
SQ5	5.5	14.3	16.5	25.3	38.5	**3.77**
SQ6	6.6	7.7	19.8	30.8	35.2	**3.80**
SQ7	13.2	13.2	11.0	26.4	36.3	**3.59**
SQ8	4.4	13.2	14.3	37.4	30.8	**3.77**

Considering the mean result, students highlighted that the innovation raised awareness about the importance of software testing to detect defects and of building quality software. In the same line, they considered important applying both static and dynamic analysis for a more complete evaluation. This means that students were able to distinguish different issues in their code when applying check programs and when executing the test suite. This result supports Naik and Tripathy's statement that dynamic analysis is not an alternative to static analysis [7] and therefore both kinds of tests should be carried out (see Section 2.1 for further information).

Regarding the open-ended question SQ9, there was a wide variety of opinions and suggestions for improvement. It was curious to read that, while some students viewed the information provided by the error/suggestion messages as scarce, other students preferred limited information as that challenged them to investigate further on the issues. Interestingly, how much information the personalised messages should provide was a quite discussed matter when adding the verifications to the check programs. A similar situation was pointed out by Saikkonen et al. [22]: only 45% of their students thought that the messages displayed by Scheme-robo were adequate or good. Based on their experience, Higgins et al. [26] found that excessive information in the feedback could result in the opposite of the desired effect. As a conclusion, these messages should not be set lightly: there should be a trade-off between setting general messages –which might be incomprehensible for the student–, and detailed messages –which might easily lead to the solution. A good strategy could be giving plenty of information in the first assignment and reduce the details progressively in subsequent exercises.

6 | OBSERVED BENEFITS AND DIFFICULTIES. LESSONS LEARNED.

The application of the innovation during three years has revealed several interesting benefits for both lecturers and students, and also some limitations. We analysed different aspects related to the application of the innovation, such as preparation of check programs, response to feedback and communication between students and lecturers. The following lessons learned can be highlighted:

- **Timely feedback**: When no assessment tool is used to assist teaching, there might be a gap of several days, or even weeks, between the day when students implement their solution and the day when they receive the corrections. In that space of time, students often forgot many details about the practice and the reasons that led them to write the code in that way. In general, they did not amend their code once they received feedback because they were focused on the new assignments. The automated assessment providing instant and specific feedback to students partially solves this problem, as they can use it to improve their solutions whenever and wherever they are addressing the task. In this way, a feedback message becomes actually effective.
- **Reduction of time spent on corrections**: Thanks to these programs, lecturers can spend less time to review students' programs because part of that correction has been previously automatised. Additionally, there are fewer issues in those solutions as the timely feedback results in better solutions in general, which is in line with the observations by Lazar et al. [3]. Indeed, we noticed that, as the course progressed, the questions about the most basic verifications were less frequent. This may mean that they also embraced little by little the guidelines and the programming style that we promoted in class or, at least, that they knew the reasons behind those checks. Sometimes, our students were even able to help each other when one of the already discussed checks appeared again at a later stage. This also relates to the study by Falkner et al. [46], who found that students tended to work harder to solve exercises as the amount of feedback increased. Finally, automated assessment reduces the chance of human error if implemented properly, and it is completely impartial as it reaches everyone equally.
- **Improvement of lecturer-student interaction**: On the one hand, check programs provide students with information related to their solutions. Based on the messages, some students modified slightly their code several times to observe correct and wrong situations. Therefore, these programs offered them the opportunity to build interaction with the lecturer upon the error/suggestion messages. For instance, the following question was quite usual "*I have noticed that, if I remove this element from this class, the error/suggestion message disappears, but I do not understand why.*"; such questions gave rise to explanations on those aspects during the classroom sessions which, otherwise, might have not been perceived as important issues by the students. On the other hand, the lecturer becomes aware of the parts of the code that cause confusion. This information can be cleverly used to reinforce explanations on those troublesome points.
- **Preparation time** The lecturer needs to dedicate time before the start of the practices. Namely, the lecturer has to:
  1. Extract the requirements that students are expected to satisfy.
  2. Implement the check program and new verifications in case that they are not implemented in the library yet.
  3. Rewrite the statement to make clear what are the requirements, and whether some elements must be declared with a particular name (see Section 4.2). It is also important to align the personalised messages to the statement so that students can effectively address their mistakes.

  The effort required to create new exercises with assessment tools is a concern [11]. As it was perceived by instructors in similar experiences with automated tools [4], the time spent initially in this process is, however, highly compensated with the benefits achieved.

- **Response to feedback:** We provided students with the check programs as a mechanism to guide them when facing the assignments. They could use them on their own whenever they needed. As a consequence, we do not have specific data about the actual students' response to the automated feedback, such as how many attempts they required to reach a correct solution. In contrast, the results of the student opinion surveys, and also our experience during these three years, tell us that our approach may be beneficial in the sense that students consider these tools as a real aid and not as a judge that can have an impact on their marks. The possibility to restrict the area of application of each verification may also have contributed to this perception; the information provided by our check programs was usually not as overwhelming as if the checks had analysed the whole code; also, the messages could be more accurate in guiding the students to the class or function they should revise.

7 | CONCLUSION AND FUTURE WORK

The approach presented in this work seeks to make accessible to lecturers the application of customisable verifications to identify inefficient and buggy fragments of code and promote that students follow good programming habits. The results of the application of the presented approach show that providing specific feedback leads to a positive overall effect. These check programs warn students that not all implementations are equally valid, helping them identify convenient solutions, and how different alternatives affect efficiency, readability and maintainability. Static and dynamic analysis have different goals, and they can therefore complement each other for more complete feedback.

The main strengths of the library are: it is intuitive to apply and extend, it can be used to assess specific situations, and it relies on mature technology that guarantees complete coverage of the language features. We have focused on academic purposes but this approach can be extrapolated to industry. In that case, the goal is that developers comply with some coding standards or guidelines defined in the company in order to homogenise and increase the quality of the code developed by all members in a project team. Also, further leverage could be gained if the library is uploaded to a collaborative system where all lecturers can contribute.

In the future, we plan to transfer the use of the library to similar subjects and evaluate the effect of its application at different stages of the degree and the evolution of this effect over time. The number of lecturers that have utilised the library is still modest. As such, no insightful assessment has been provided on how difficult the library is to use or how useful it is as a teaching tool. Therefore, we also aim to analyse the usability of the library once its application has spread to similar courses. To this end, we will conduct some experiments to observe the ability of a number of lecturers, with different levels of expertise, when acting as users and developers of the library. Another idea for the future is to integrate the library into the virtual platform of the course so that students can upload their solutions and receive feedback. In this way, we will be able to collect statistics about the number of attempts before reaching a correct solution. Additionally, by analysing the most launched verifications, we could better understand the points in which students are prone to mistakes. Furthermore, it would be possible to issue challenges (e.g., reach a solution using the fewest possible number of conditional statements) to instil the importance of the software quality factors in the students.

Acknowledgment

The authors would like to thank Gerardo Aburruzaga García, Jose Fidel Argudo Argudo, Francisco Orrequia Terrero, Daniel Pérez Caro and Fernando Manuel Quintana Velázquez, who contributed to the development and testing of the CAC++ library.

references

[1] Kalogeropoulos N, Tzigounakis I, Pavlatou EA, Boudouvis AG. Computer-based assessment of student performance in programing courses. Comput Appl Eng Educ 2013;21(4):671–683. https://dx.doi.org/10.1002/cae.20512.

[2] Charman D, Elmes A. Computer Based Assessment: A Guide to Good Practice, vol. I. University of Plymouth; 1998.

[3] Lazar T, Sadikov A, Bratko I. Rewrite Rules for Debugging Student Programs in Programming Tutors. IEEE Transactions on Learning Technologies 2018 Oct;11(4):429–440.

[4] Pettit RS, Homer JD, Holcomb KM, Simone N, Mengel SA. Are Automated Assessment Tools Helpful in Programming Courses? In: 2015 ASEE Annual Conference & Exposition Seattle, Washington: ASEE Conferences; 2015. http://dx.doi.org/10.18260/p.23569.

[5] Striewe M, Goedicke M. A Review of Static Analysis Approaches for Programming Exercises. In: Kalz M, Ras E, editors. Computer Assisted Assessment. Research into E-Assessment Cham: Springer International Publishing; 2014. p. 100–113.

[6] Howatt JW. On Criteria for Grading Student Programs. SIGCSE Bull 1994 Sep;26(3):3–7. http://dx.doi.org/10.1145/187387.187389.

[7] Naik S, Tripathy P. Software Testing and Quality Assurance: Theory and Practice. Wiley-Spektrum; 2008.

[8] Horstmann CS, Budd TA. Big C++, 2nd Edition. Wiley; 2009.

[9] Penttilä E, Improving C++ Software Quality with Static Code Analysis; 2014.

[10] Ala-Mutka K, Uimonen T, Jarvinen HM. Supporting students in C++ programming courses with automatic program style assessment. J Inform Technol Educ 2004;3(1):245–262. http://dx.doi.org/10.28945/300.

[11] Keuning H, Jeuring J, Heeren B. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. ACM Trans Comput Educ 2018 Sep;19(1):3:1–3:43. http://dx.doi.org/10.1145/3231711.

[12] Clang: a C language family frontend for LLVM; 2018. http://clang.llvm.org, accessed 30 July 2019.

[13] Jones J. Abstract syntax tree implementation idioms. In: Proceedings of the 10th Conference on Pattern Languages of Programs (PLOP 2003); 2003. p. 1–10.

[14] Goedicke M, Striewe M, Balz M. Computer aided assessments and programming exercises with JACK. University Duisburg-Essen, Institute for Computer Science and Business Information Systems (ICB); 2008.

[15] Ala-Mutka KM. A Survey of Automated Assessment Approaches for Programming Assignments. Comput Sci Educ 2005;15(2):83–102. https://dx.doi.org/10.1080/08993400500150747.

[16] Douce C, Livingstone D, Orwell J. Automatic Test-based Assessment of Programming: A Review. J Educ Resour Comput 2005 Sep;5(3). http://dx.doi.org/10.1145/1163405.1163409.

[17] Benford S, Burke E, Foxley E, Gutteridge N, Zin AM. Ceilidh as a Course Management Support System. J Educ Technol Sys 1994;22(3):235–250. https://dx.doi.org/10.2190/Y62D-CYEX-Q4D8-R8DQ.

[18] Striewe M, Balz M, Goedicke M. A Flexible and Modular Software Architecture for Computer Aided Assessments and Automated Marking. In: CSEDU 2009 - Proceedings of the First International Conference on Computer Supported Education, Lisboa, Portugal, March 23-26, 2009 - Volume 2; 2009. p. 54–61.

[19] Benford S, Burke E, Foxley E, Gutteridge N, Zin AM. Early experiences of computer-aided assessment and administration when teaching computer programming. Res Learn Technol 1993;1(2):55–70. https://dx.doi.org/10.1080/0968776930010206.

[20] Jackson D, Usher M. Grading Student Programs Using ASSYST. In: Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education SIGCSE '97, New York, NY, USA: ACM; 1997. p. 335–339. http://dx.doi.org/10.1145/268084.268210.

[21] Schorsch T. CAP: An Automated Self-assessment Tool to Check Pascal Programs for Syntax, Logic and Style Errors. In: Proceedings of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education SIGCSE '95, New York, NY, USA: ACM; 1995. p. 168–172. http://dx.doi.org/10.1145/199688.199769.

[22] Saikkonen R, Malmi L, Korhonen A. Fully Automatic Assessment of Programming Exercises. SIGCSE Bull 2001 Jun;33(3):133–136. http://dx.doi.org/10.1145/507758.377666.

[23] Solomon A, Santamaria D, Lister R. Automated Testing of Unix Command-line and Scripting Skills. In: 7th International Conference on Information Technology Based Higher Education and Training (ITHET'06); 2006. p. 120–125.

[24] Fu X, Peltsverger B, Qian K, Tao L, Liu J. APOGEE: Automated Project Grading and Instant Feedback System for Web Based Computing. SIGCSE Bull 2008 Mar;40(1):77–81. http://dx.doi.org/10.1145/1352322.1352163.

[25] Ahmadzadeh M, Namvar S, Soltani M. JavaMarker: A Marking System for Java Programs. Int J Comput Appl 2011 April;20(2):15–20. http://dx.doi.org/10.5120/2407-3202.

[26] Higgins C, Hegazy T, Symeonidis P, Tsintsifas A. The CourseMarker CBA System: Improvements over Ceilidh. Educ Inform Technol 2003 Sep;8(3):287–304. https://dx.doi.org/10.1023/A:1026364126982.

[27] Joy M, Griffiths N, Boyatt R. The BOSS Online Submission and Assessment System. J Educ Resour Comput 2005;5(3). http://dx.doi.org/10.1145/1163405.1163407.

[28] Meyers S. Effective C++: 50 Specific Ways to Improve Your Programs and Designs. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc.; 1992.

[29] Henricson M, Nyquist E, Programming in C++, rules and recommendations. Ellemtel Telecommunication Systems Laboratories; 1992.

[30] Dromey RG. A model for software product quality. IEEE T Software Eng 1995 Feb;21(2):146–162. http://dx.doi.org/10.1109/32.345830.

[31] Papancea A, Spacco J, Hovemeyer D. An Open Platform for Managing Short Programming Exercises. In: Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research ICER '13, New York, NY, USA: ACM; 2013. p. 47–52. http://dx.doi.org/10.1145/2493394.2493401.

[32] Zanden BV, Anderson D, Taylor C, Davis W, Berry MW. CodeAssessor: An Interactive, Web-based Tool for Introductory Programming. J Comput Sci Coll 2012 Dec;28(2):73–80.

[33] PMD, An extensible cross-language static code analyzer; 2018. https://pmd.github.io/, accessed 30 July 2019.

[34] SonarQube, Continuous Code Quality; 2018. https://www.sonarqube.org/, accessed 30 July 2019.

[35] Jackson D. A Semi-automated Approach to Online Assessment. SIGCSE Bull 2000 Jul;32(3):164–167. http://dx.doi.org/10.1145/353519.343160.

[36] Laakso MJ, Salakoski T, Korhonen A. The Feasibility of Automatic Assessment and Feedback. In: Cognition and Exploratory Learning in Digital Age IEEE Technical Committee on Learning Technology and Japanese Society of Information and Systems in Education; 2005. p. 113–122.

[37] Restrepo-Calle F, Ramírez Echeverry JJ, González FA. Continuous assessment in a computer programming course supported by a software tool. Comput Appl in Eng Educ 2019;27(1):80–89. https://dx.doi.org/10.1002/cae.22058.

[38] Clarke PJ, Davis D, King TM, Pava J, Jones EL. Integrating Testing into Software Engineering Courses Supported by a Collaborative Learning Environment. ACM T Comput Educ 2014 Oct;14(3):18:1–18:33. http://dx.doi.org/10.1145/2648787.

[39] Clegg BS, Rojas JM, Fraser G. Teaching Software Testing Concepts Using a Mutation Testing Game. In: Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education Track ICSE-SEET '17, Piscataway, NJ, USA: IEEE Press; 2017. p. 33–36. https://dx.doi.org/10.1109/ICSE-SEET.2017.1.

[40] Lemos OAL, Silveira FF, Ferrari FC, Garcia A. The impact of Software Testing education on code reliability: An empirical assessment. J Syst Software 2018;137:497 – 511. https://dx.doi.org/10.1016/j.jss.2017.02.042.

[41] ISO. ISO/IEC 14882:2011 Information technology – Programming languages – C++. Geneva, Switzerland: International Organization for Standardization; 2012.

[42] Delgado-Pérez P, Medina-Bulo I, Palomo-Lozano F, García-Domínguez A, Domínguez-Jiménez JJ. Assessment of class mutation operators for C++ with the MuCPP mutation system. Inform Software Technol 2017;81:169–184. http://dx.doi.org/10.1016/j.infsof.2016.07.002.

[43] Jia Y, Harman M. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In: Testing: Academic Industrial Conference - Practice and Research Techniques (TAIC PART 2008); 2008. p. 94–98. http://dx.doi.org/10.1109/TAIC-PART.2008.18.

[44] AST Matcher Reference; 2018. http://clang.llvm.org/docs/LibASTMatchersReference, accessed 30 July 2019.

[45] Neamtiu I, Foster JS, Hicks M. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. In: Proceedings of the 2005 International Workshop on Mining Software Repositories MSR '05, New York, NY, USA: ACM; 2005. p. 1–5. http://dx.doi.org/10.1145/1082983.1083143.

[46] Falkner N, Vivian R, Piper D, Falkner K. Increasing the Effectiveness of Automated Assessment by Increasing Marking Granularity and Feedback Units. In: Proceedings of the Forty-fifth ACM Technical Symposium on Computer Science Education SIGCSE '14, New York, NY, USA: ACM; 2014. p. 9–14. http://dx.doi.org/10.1145/2538862.2538896.